Facing the Reality of Data Stream Classification: Coping with Scarcity of Labeled Data

Mohammad Mehedy Masud[1], Jing Gao[2], Latifur Khan[1]

Jiawei Han[2] and Bhavani Thuraisingham[1]

[1]Department of Computer Science, University of Texas at Dallas

[2]Department of Computer Science, University of Illinois at Urbana Champaign

Abstract. Recent approaches in classifying data streams are based on supervised learning algorithms, which can be trained with labeled data only. Manual labeling of data is both costly and time consuming. Therefore, in a real streaming environment, where large volume of data appear at a high speed, only a small fraction of the data can be labeled. Thus, only a limited number of instances will be available for training/updating the classification models, leading to poorly trained classifiers. We apply a novel technique to overcome this problem by utilizing both unlabeled and labeled instances to train/update the classification model. Each classification model is built as a collection of micro-clusters using semi-supervised clustering, and an ensemble of these models is used to classify unlabeled data using nearest neighbor algorithm. Empirical evaluation on both synthetic and real data reveals that our approach, using only a small amount of labeled data for training, outperforms state-of-the-art stream classification algorithms that use five times more labeled data than our approach.

Keywords: Data stream classification, semi-supervised clustering, ensemble classification, concept-drift.

1. Introduction

Data stream classification is a challenging problem because of two important properties of the stream: its infinite length and evolving nature. Data streams evolve when the joint probability distribution $p(c, x) = p(c)p(x|c)$ changes over time, where c denotes class label and x denotes feature vector. So, evolution occurs under three circumstances. First, the prior probability distribution $p(c)$ changes. This happens if a new class emerges in the stream (concept-evolution). Second, the conditional probability $p(x|c)$ changes as a result of changes in the underlying concept of the data (concept-drift). Finally, both $p(c)$ and $p(x|c)$

change. In any case, the challenge is to build a classification model that is consistent with the current concept. Almost all of the existing data stream classification techniques [1, 7, 12, 14, 18, 21, 24, 29, 31] are based on an impractical assumption that the true label of a data point becomes available immediately after it is tested by the classifier. In other words, the data stream is assumed to be "completely labeled", meaning, the true labels of all historical data are known. This assumption is impractical because manual labeling of data is usually costly and time consuming. So, in an streaming environment, where data appear at a high speed, it is not possible to manually label all the data as soon as they arrive. If it were possible, we would not need a classifier to begin with. Thus, in practice, only a small fraction of the stream can be labeled by human experts. So, the traditional stream classification algorithms would have very few instances to update their model, leading to a poorly built classifier.

As an example, suppose an organization receives flight reports as text documents from all over the world, at a rate of thousand reports per day, and categorizes the reports into different classes: "normal", "minor mechanical problem", "minor weather problem", "major mechanical problem", ... , and so on. Based on the categories, warning messages are sent to the corresponding airlines and aviation authorities for proper action. Important decision-making actions such as flight planning, resource allocation, and personnel assignment are affected by these warnings. Therefore, timely delivery of these warnings is necessary to avoid both financial loss, and customer dissatisfaction. Without loss of generality, suppose only 200 of the reports can be labeled manually by human experts each day. So, an automated stream document classification system is employed so that all 1,000 documents can be classified each day. If a traditional stream classification technique is used, it will have to deal with a trade-off when updating the classifier. Either the classifier will have to be updated with only 200 labeled data per day, or it will have to wait 5 days to be updated with all the 1,000 labeled data that arrived today. None of the trade-offs are acceptable since the former will lead to a poor classifier, and the latter will lead to an outdated classifier. In order to completely avoid these problems, the organization must increase its manpower (and cost) 5 times and classify all the 1,000 instances manually.

Considering these difficulties, we propose an algorithm that updates the existing classification model utilizing the available 200 labeled and 800 unlabeled instances, while achieving the same or better classification accuracy than a classification model that is updated using 1,000 labeled instances. Thus, our approach offers a practical data stream classifier that not only views the data stream classification problem from a real perspective, but also provides a cost-effective solution. Thus, our algorithm is capable of building efficient classification models with a "partially labeled" data stream, compared to other stream classification techniques that require "completely labeled" stream. By "partially labeled" we mean only a fraction (e.g. 20%) of the instances in the data stream are labeled, and by "completely labeled" we mean all (100%) the instances are labeled.

Naturally, stream data could be stored in buffer and processed when the buffer is full, so we divide the stream data into equal sized chunks. We train a classification model from each chunk. We propose a semi-supervised clustering algorithm to create K clusters from the partially labeled training data [22]. A summary of the statistics of the instances belonging to each cluster is saved as a "micro-cluster". The micro-clusters created from each chunk serve as a classification model for the nearest neighbor algorithm. In order to cope with concept-drift, we keep an ensemble of L models. Whenever a new model is built from a new data chunk, we update the ensemble by choosing the best L models from the $L+1$ models (previous L models and the new model), based on their individual accuracies on the labeled training data of the new data chunk. Besides, we refine the existing models in the ensemble whenever a new class of data evolves in the stream.

We have several contributions. First, we propose an efficient semi-supervised clustering algorithm based on cluster-impurity measure. Second, we apply our technique to classify evolving data streams. To the best of our knowledge, there are no stream data classification algorithms that apply semi-supervised clustering. Third, we provide a solution to the more practical situation of stream classification when labeled data are scarce. We show that our approach, using only 20% labeled training data, achieves better classification accuracy in real life data sets than other stream classification approaches that use 100% labeled training data. We believe that the proposed method provides a promising, powerful, and practical technique to the stream classification problem in general.

The rest of the paper is organized as follows: section 2 discusses related work, section 3 provides an overview of the classification process, section 4 describes the theoretical background and the implementation of the semi-supervised clustering, section 5 discusses the ensemble classification and ensemble updating process with micro-clusters, section 6 discusses data set and experimental setup, and evaluation of our approach, section 7 discusses our findings and section 8 concludes with directions to future work.

2. Related work

Our work is related to both semi-supervised clustering and stream classification techniques. We briefly discuss both of them.

Semi-supervised clustering techniques utilize a small amount of knowledge available in the form of pairwise constraints (*must-link, cannot-link*), or class labels of the data points. According to [4], semi-supervised clustering techniques can be subdivided into two categories: constraint-based and distance-based. Constraint-based approaches, such as [2, 9, 28] try to cluster the data points without violating the given constraints. Distance-based techniques use a specific distance metric or similarity measure (e.g. Euclidean distance), but the distance metric is parameterized so that it can be adjusted to satisfy the given constraints. Examples of the distance-based techniques are [8, 16, 20, 30]. Some recent approaches for semi-supervised clustering integrated the search-based and constraint-based techniques into a unified framework, by applying pairwise constraints on top of the unsupervised K-means clustering technique and formulating a constrained K-means clustering problem ([3, 4, 6]). These approaches usually apply the Expectation-Maximization (E-M) technique to solve the constrained clustering problem.

Our approach follows the constraint-based technique, but it is different from other constraint-based approaches. Most constraint-based approaches use pair-wise constraints (e.g. [6], whereas we utilize a cluster-impurity measure based on the limited labeled data contained in each cluster [22]. If pair-wise constraints are used, then the running time per E-M step is quadratic in total number of labeled points, whereas the running time is linear if impurity measures are used. So, the impurity measures are more realistic in classifying a high-speed stream data. Although Basu et al. [2] did not use any pair-wise constraints, they did not use any cluster-impurity measure either. However, Cluster-impurity measure was used by Demiriz et al [9]. But they applied expensive genetic algorithms, and had to adjust weights given to different components of the clustering objective function to obtain good clusters. On the contrary, we apply E-M, and we do not need to tune parameters to get a better objective function. Furthermore, we use a compound impurity-measure rather than the simple impurity-measures used in [9]. Besides, to the best of our knowledge, no other work applies a semi-supervised clustering technique to classify stream data.

There have been many works in stream data classification. There are two main approaches - single model classification, and ensemble classification. Some single model classification techniques incrementally update their model when new data arrives [15,26]. However, these techniques apply costly operations to update the internal structure of the model, which is not suitable for a high-speed data stream. Domingos and Hulten [11] propose a single model incremental algorithm that can cope with the speed of the stream. However, these techniques are not capable of handling concept-drift, which occurs frequently in data streams. Hulten et al. [18] propose an incremental decision tree that can handle concept-drift as well as high-speed of the data stream. However, in any incremental algorithm, only the most recent data is used to update the model. Thus, contributions of historical data are forgotten at a constant rate even if some of the historical data are consistent with the current concept. So, the refined model may not appropriately reflect the current concept, and its prediction accuracy may not meet the expectation.

In a non-streaming environment, ensemble classifiers like Boosting [13] are popular alternatives to single model classifiers. But these are not directly applicable to stream mining since they require multiple passes over the entire training data, which is impractical for in an streaming environment. However, several ensemble techniques for stream data mining have been proposed [12, 14, 21, 24, 29]. These ensemble approaches have the advantage that they can be more efficiently built than updating a single model and they observe higher accuracy than their single model counterpart [25].

Our approach is also an ensemble approach, but it is different from other ensemble approaches in two aspects. First, previous ensemble-based techniques use the underlying learning algorithm (such as decision tree, Naive Bayes, etc.) as a black-box and concentrate only on optimizing the ensemble. But we concentrate mainly on building efficient classification models in an evolving scenario. In this light, our work is more closely related with the work of Aggarwal et al [1]. Secondly, previous techniques such as [1] require *completely labeled* training data. But in practice, a very limited amount of labeled data may be available in the stream, leading to poorly trained classification models. We show that high classification accuracy can be achieved even with limited amount of labeled data.

Aggarwal et al. [1] apply a supervised micro-clustering technique along with horizon-fitting to classify evolving data streams. They have achieved higher accuracy than other approaches that use fixed horizon or the entire dataset for training. We also apply a micro-clustering technique. But there are two major differences between our approach and this approach. First, we do not use horizon-fitting for classification. Rather, we use a fixed-sized ensemble of classifiers. So, we do not need to store historical snapshots, which allows us to save memory. Second, we apply semi-supervised clustering, rather than supervised. Thus, we need only a fraction of training data to be labeled, compared to a *completely labeled* data that is required for the previous approach. Thus, our approach not

Table 1. Symbols and terms

S : Chunk size	K: Number of micro-clusters
L : Ensemble size	P : Percentage of labeled data in each chunk
D_i : A data chunk	C : Number of classes in the stream
M : The ensemble	M^i : The i-th model in the ensemble

Labeled instance: An instance that has been correctly labeled by an independent labeling mechanism (e.g. domain expert)
Partially labeled chunk: A data chunk having at least P% labeled instances
Completely labeled chunk: A data chunk having 100% labeled instances

only saves more memory, but also it is more applicable to a realistic scenario where labeled data are scarce.

Our current work is an extension to the previous work [22]. In the previous work, it was assumed that there were two parallel, disjoint streams: a training stream and a test stream. The training stream contained the labeled instances, and was used to train the models. The test stream contained the unlabeled instances and was used for testing. However, this assumption was not so realistic since in a real world scenario, labeled data may not be immediately available in the stream, and therefore, it may not be possible to construct a separate training stream. So, in this paper, we make a more realistic assumption that there is a single continuous stream. Each data chunk in the stream is first tested by the existing ensemble, and then the same chunk is used for training, assuming that the instances in the chunk have been labeled. Thus, all the instances in the stream are eventually tested by the ensemble. Besides, in this paper, we have described our technique more elaborately and provided detailed understanding and proof of the proposed framework. Finally, we have enriched the experimental results by adding three more datasets, run more rigorous experiments, and reported in-depth analyses of the results.

3. Top level description

At first, we informally define the data stream classification problem. We assume that data arrive in chunks, as follows:

$$\begin{aligned} D_1 &= x_1, ..., x_S \\ D_2 &= x_{S+1}, ..., x_{2S} \\ &... \\ &... \\ D_n &= x_{(n-1)S+1}, ..., x_{nS} \end{aligned}$$

where x_i is the i-th instance in the stream, S is the chunk size, D_i is the i-the data chunk, and D_n is the latest data chunk. Assuming that the class labels of all the instances in D_n are unknown, the problem is to predict their class labels. Let y_i and $\hat{y_i}$ be the actual and predicted class labels of x_i, respectively. If $\hat{y_i}$=y_i, then the prediction is correct, otherwise it is incorrect. The goal is to minimize the prediction error.

Table 1 explains the terms and symbols that are used throughout the paper. Our approach will be referred to henceforth as "ReaSC", which stands for "Realistic Stream Classifier". Figure 1 shows the top level architecture of ReaSC.

We train a classification model from a data chunk D_i as soon as P% ($P <<$

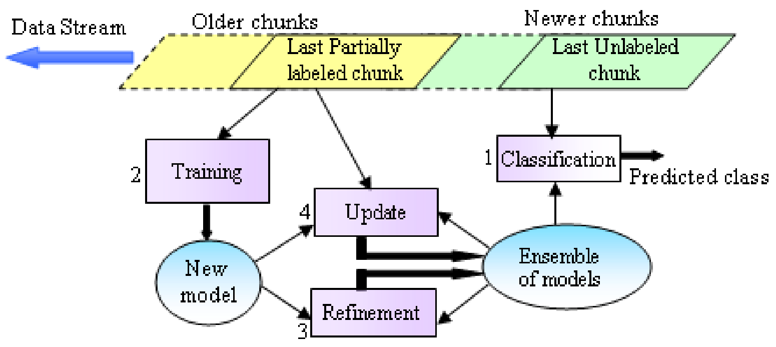


Fig. 1. Overview of ReaSC

100) randomly chosen instances from the chunk have been correctly labeled by an independent labeling mechanism (e.g., human experts). Note that this assumption is less strict than other stream classification techniques such as [29], which assumes that all the instances of D_i must have been labeled before it can be used to train a model. We build the initial ensemble M of L models = $\{M^1, ..., M^L\}$ from the first L data chunks, where M^i is trained from chunk D_i. Then the following algorithm is applied for each of the following chunk.

Algorithm 1 ReaSC

```
Input: $D_n$: Latest data chunk
    $M$: current ensemble of $L$ models $\{M^1, ..., M^L\}$
Output: Updated ensemble $M$
1: for all $x_i \in D_n$ do $\hat{y}_i \leftarrow$ Classify($M$,$x_i$) (section 5.1)
   /* Assuming that $P\%$ instances in $D_n$ has now been labeled */
2: $M' \leftarrow$ Train($D_n$) (section 4) /* Build a new model $M'$ */
3: $M \leftarrow$ Refine-Ensemble($M, M'$) (section 5.2)
4: $M \leftarrow$ Update-Ensemble($M, M', D_n$) (section 5.3)
5: return $M$
```

The main steps of algorithm 1 (ReaSC) are explained below.

1. Classification: The existing ensemble is used to predict the labels of each instance in D_n using Nearest Neighbor (NN) classification, and majority voting (section 5.1). As soon as D_n has been partially labeled, the following steps are performed.

2. Training: Training is done by applying semi-supervised clustering on the partially-labeled training data to build K clusters (section 4). The semi-supervised clustering is based on the Expectation-Maximization(E-M) algorithm that locally minimizes an objective function. The objective function takes into account the dispersion between each point and its corresponding cluster centroid, as well as the impurity-measure of each cluster. Then we extract a statistical summary from the data points of each cluster, save the summary as a micro-cluster, and remove the raw data points (section 4.4). In this way, we get a new classification model M' that can be used to classify unlabeled data using the nearest neighbor (NN) algorithm.

3. Ensemble refinement: In this step M' is used to refine the existing ensemble of models if required (section 5.2). Refinement is required if M' contains

Table 2. An example of ReaSC actions with stream progression

⇓ Stream progression	Arrival of chunk	Action(s)
	D_1	—
	D_2	$M^1 \leftarrow$ Train(D_1)
	...	...
	...	...
	D_{L+1}	$M^L \leftarrow$ Train(D_L), Initial model $M = \{M^1, ..., M^L\}$
		$\forall x_j \in D_{L+1}\ \hat{y}_j \leftarrow$ Classifiy(M,x_j)
	D_{L+2}	$M' \leftarrow$ Train(D_{L+1})
		$M \leftarrow$ Refine-Ensemble(M,M')
		$M \leftarrow$ Update-Ensemble(M,M',D_{L+1})
		$\forall x_j \in D_{L+2}\ \hat{y}_j \leftarrow$ Classifiy(M,x_j)
	...	...
	...	...
	D_{L+i}	$M' \leftarrow$ Train(D_{L+i-1})
		$M \leftarrow$ Refine-Ensemble(M,M')
		$M \leftarrow$ Update-Ensemble(M,M',D_{L+i-1})
		$\forall x_j \in D_{L+i}\ \hat{y}_j \leftarrow$ Classifiy(M,x_j)
	...	...
	...	...

some data of a particular class c, but no model in the ensemble M contains any data of that class. This situation may occur because of concept-evolution. In this case, the existing ensemble M does not have any knowledge of class c, and so, it must be refined so that it learns to classify instances of this class. Refinement is done by injecting micro-clusters of M', which contain labeled instances of class c, into the existing models of the ensemble.

4. Ensemble update: In this step, we select the best L models from the L+1 models: $M \cup \{M'\}$, based on their accuracies on the labeled instances of D_n (section 5.3). These L best models construct the new ensemble M. The ensemble technique helps the system to cope with concept-drift.

Table 2 illustrates an schematic example of ReaSC. In this example, we assume that P% data in data chunk D_i are labeled by the time chunk D_{i+1} arrives. The initial ensemble is built with the first L chunks. Then the ensemble is used to classify the latest chunk (D_{L+1}). From the next (L+2nd) chunk onward, a sequence of operations are performed with the arrival of a new chunk. For example, the sequence of operations at the arrival of chunk D_{L+i} ($i > 1$) is as follows:
i) The previous chunk D_{L+i-1} has been partially labeled by now. Train a new model M' using D_{L+i-1}.
ii) Refine the existing ensemble M using the new model M'.
iii) Update the ensemble M by choosing the best L models from $M \cup \{M'\}$.
iv) Classify each instance in D_{L+i} using ensemble M.

4. Training with limited labeled data

As mentioned earlier, we train a classification model from each partially labeled data chunk. The classification model is a collection of K micro-clusters obtained using semi-supervised clustering. Training consists of two basic steps: semi-supervised clustering, and storing the cluster summaries as micro-clusters.

In the semi-supervised clustering problem, we are given a set of m data points $\mathcal{X} = \{\boldsymbol{x_1}, ..., \boldsymbol{x_l}, \boldsymbol{x_{l+1}}, ..., \boldsymbol{x_m}\}$, where the first l instances are labeled, i.e., $y_i \in \{1, ..., C\}$, $i \leq l$, and the remaining instances are unlabeled; C being the total number of classes. We assign the class label y_i=0 for all unlabeled instance x_i, $i > l$. We are to create K clusters, maintaining the constraint that all points in the same cluster have the same class label. We restrict the value of parameter K to be greater than C, since intuitively, there should be at least one cluster for each class of data. We'll first re-examine the unsupervised K-means clustering in section 4.1 and then propose a new semi-supervised clustering technique using cluster-impurity minimization in section 4.2.

4.1. Unsupervised K-means clustering

The unsupervised K-means clustering creates K-partitions of the data points based on the only available information, the similarity/dispersion measure among the data points. The objective is to minimize the sum of dispersion between each data point and its corresponding cluster centroid (i.e., intra-cluster dispersion). Given m unlabeled data points $\mathcal{X} = \{\boldsymbol{x_1}, \boldsymbol{x_2}, ..., \boldsymbol{x_m}\}$, K-means creates K-partitions $\{\mathcal{X}_1, ..., \mathcal{X}_K\}$ of $\mathcal{X}$, minimizing the objective function:

$$\mathcal{O}_{Kmeans} = \sum_{i=1}^{K} \sum_{\boldsymbol{x} \in \mathcal{X}_i} ||\boldsymbol{x} - \boldsymbol{u_i}||^2 \tag{1}$$

where $\boldsymbol{u_i}$ is the centroid of cluster i, and $||\boldsymbol{x} - \boldsymbol{u_i}||$ is the Eucledian distance between $\boldsymbol{x}$ and $\boldsymbol{u_i}$.

4.2. K-means clustering with cluster-impurity minimization

Given a limited amount of labeled data, the goal for K-means with Minimization of Cluster Impurity (MCI-Kmeans) is to minimize the intra-cluster dispersion (same as unsupervised K-means) and at the same time minimize the impurity of each cluster. A cluster is completely pure if it contains only unlabeled instances, or labeled instances from only one class. Thus, the objective function should penalize each cluster for being impure. The general form of the objective function is as follows:

$$\mathcal{O}_{MCIKmeans} = \sum_{i=1}^{K} \sum_{\boldsymbol{x} \in \mathcal{X}_i} ||\boldsymbol{x} - \boldsymbol{u_i}||^2 + \sum_{i=1}^{K} \mathcal{W}_i * Imp_i \tag{2}$$

where $\mathcal{W}_i$ is the weight associated with cluster i and Imp_i is the impurity of cluster i. In order to ensure that both the intra-cluster dispersion and cluster impurity are given the same importance, the weight associated with each cluster should be adjusted properly. Besides, we would want to penalize each data point that contributes to the impurity of the cluster. So, the weight associated with each cluster is chosen to be

$$\mathcal{W}_i = |\mathcal{X}_i| * \bar{D}_{\mathcal{X}_i} \tag{3}$$

where $\mathcal{X}_i$ is the set of data points in cluster i and $\bar{D}_{\mathcal{X}_i}$ is the average dispersion of each of these points from the cluster centroid. Thus, each instance has a contribution to the total penalty, which is equal to the cluster impurity multiplied by the average dispersion of the data points from the centroid. We observe that equation (3) is equivalent to the sum of dispersions of all the instances from the cluster centroid. That is, we may rewrite equation (3) as:

$$\mathcal{W}_i = \sum_{\boldsymbol{x} \in \mathcal{X}_i} ||\boldsymbol{x} - \boldsymbol{u_i}||^2$$

Substituting this value of $\mathcal{W}_i$ in equation (2) we obtain:

$$\mathcal{O}_{MCIKmeans} = \sum_{i=1}^{K} \sum_{\boldsymbol{x} \in \mathcal{X}_i} ||\boldsymbol{x} - \boldsymbol{u_i}||^2 + \sum_{i=1}^{K} \sum_{\boldsymbol{x} \in \mathcal{X}_i} ||\boldsymbol{x} - \boldsymbol{u_i}||^2 * Imp_i$$

$$= \sum_{i=1}^{K} (\sum_{\boldsymbol{x} \in \mathcal{X}_i} ||\boldsymbol{x} - \boldsymbol{u_i}||^2 (1 + Imp_i)) \tag{4}$$

Impurity measures: Equation (4) should be applicable to any impurity measure in general. Entropy and Gini index are most commonly used impurity measures. We use the following impurity measure: $Imp_i = \mathcal{ADC}_i * Ent_i$, where $\mathcal{ADC}_i$ is the "aggregated dissimilarity count" of cluster i and Ent_i is the entropy of cluster i. The reason for using this impurity measure will be explained shortly. In order to understand $\mathcal{ADC}_i$, we first need to define "Dissimilarity count".

Definition 1 (Dissimilarity count). Dissimilarity count $\mathcal{DC}_i(\boldsymbol{x}, y)$ of a data point $\boldsymbol{x}$ in cluster i having class label y is the total number of instances in that cluster having class label other than y.

In other words,

$$\mathcal{DC}_i(\boldsymbol{x}, y) = |\mathcal{X}_i| - |\mathcal{X}_i(y)| \tag{5}$$

where $\mathcal{X}_i(y)$ is the set of instances in cluster i having class label $= y$. Recall that unlabeled instances are assumed to have class label $= 0$. Note that $\mathcal{DC}_i(\boldsymbol{x}, y)$ can be computed in constant time, if we keep an integer vector to store the counts $|\mathcal{X}_i(c)|, c \in \{0, 1, .., C\}$. "Aggregated dissimilarity count" or $\mathcal{ADC}_i$ is the sum of the dissimilarity counts of all the points in cluster i:

$$\mathcal{ADC}_i = \sum_{x \in \mathcal{X}_i} DC_i(\boldsymbol{x}, y). \tag{6}$$

Entropy of a cluster i is computed as:

$$Ent_i = \sum_{c=0}^{C} (-p_c^i * log(p_c^i))$$

where p_c^i is the prior probability of class c, i.e.,

$$p_c^i = \frac{|\mathcal{X}_i(c)|}{|\mathcal{X}_i|}. \tag{7}$$

The use of Ent_i in the objective function ensures that clusters with higher entropy get higher penalties. However, if only Ent_i had been used as the impurity measure, then each point in the same cluster would have received the same penalty. But we would like to favor the points belonging to the majority class in a cluster, and disfavor the points belonging to the minority classes. Doing so would force more points of the majority class to be moved into the cluster, and more points of the minority classes to be moved out of the cluster, making the clusters purer. This is ensured by introducing $\mathcal{ADC}_i$ to the equation. We call the combination of $\mathcal{ADC}_i$ and Ent_i as "compound impurity measure" since it can be shown that $\mathcal{ADC}_i$ is proportional to the "gini index" of cluster i. Following from equation (6), we obtain

$$\begin{aligned}
\mathcal{ADC}_i &= \sum_{x\in\mathcal{X}_i} DC_i(\boldsymbol{x}, y) = \sum_{c=0}^{C}\sum_{x\in\mathcal{X}_i(c)} DC_i(\boldsymbol{x}, y) \\
&= \sum_{c=0}^{C}\sum_{x\in\mathcal{X}_i(c)} (|\mathcal{X}_i| - |\mathcal{X}_i(c)|) \text{ (using equation 5)} \\
&= \sum_{c=0}^{C}(|\mathcal{X}_i(c)|)(|\mathcal{X}_i| - |\mathcal{X}_i(c)|) = (|\mathcal{X}_i|)^2\sum_{c=0}^{C}(\frac{\mathcal{X}_i(c)}{\mathcal{X}_i})(1-\frac{\mathcal{X}_i(c)}{\mathcal{X}_i}) \\
&= (|\mathcal{X}_i|)^2\sum_{c=0}^{C}(p_c^i)(1-p_c^i) \text{ (using equation 7)} \\
&= (|\mathcal{X}_i|)^2(1-\sum_{c=0}^{C}(p_c^i)^2) = (|\mathcal{X}_i|)^2 * Gini_i
\end{aligned}$$

where $Gini_i$ is the gini index of cluster i.

4.3. Optimizing the objective function with Expectation Maximization (E-M)

The problem of minimizing equation (4) is an *incomplete-data problem* because the cluster labels and the centroids are all unknown. The common solution to this problem is to apply E-M [10]. The E-M algorithm consists of three basic steps: initialization, E-step and M-step. Each of them is discussed here.

Initialization with proportionate cluster distribution: For each class c appearing in the data, we initialize $k_c \le K$ centroids by choosing k_c points from the labeled data of class c. The ratio of k_c to K is chosen to be equal to the ratio of the number of labeled points having class label c to the total number of labeled points in the dataset. That is, $k_c = K * \frac{|\mathcal{L}(c)|}{|\mathcal{L}|}$, $c \in \{1, ..., C\}$, where $\mathcal{L}$ is the set of all labeled points in $\mathcal{X}$, and $\mathcal{L}(c)$ is the subset of points in $\mathcal{L}$ belonging to class c. We observed in our experiments that this initialization works better than initializing equal number of centroids of each class. This is because if we initialize the same number of centroids from each class, then larger classes (i.e., classes having more instances) tend to create larger and sparser clusters, which leads to poorer classification accuracy for the nearest neighbor classification.

Let there be η_c labeled points of class c in the dataset. If $\eta_c > k_c$, then we choose k_c centroids from η_c points using the farthest-first traversal heuristic [17]. To apply this heuristic, we first initialize a "visited set" of points with a randomly chosen point having class label c. At each iteration, we find a point x_j of class c that maximizes the minimum distance from all points in the visited set, and add it to the visited set. This process continues until we have k_c points in the set. If $\eta_c < k_c$, then we choose remaining centroids randomly from the unlabeled points. After initialization, E-Step and M-Step are iterated until the convergence condition is fulfilled.

E-Step: In E-Step, we assign each data point $\boldsymbol{x}$ to a cluster i such that its contribution to the global objective function, $\mathcal{O}_{MCIKeans}(\boldsymbol{x})$, is minimized:

$$\mathcal{O}_{MCIKeans}(\boldsymbol{x}) = ||\boldsymbol{x} - \boldsymbol{u_i}||^2 * (1 + Ent_i * \mathcal{DC}_i(\boldsymbol{x}, y))$$

Note that the value of the global objective function $\mathcal{O}_{MCIKeans}$ depends on the order in which the labeled points are assigned to clusters. It is computationally intractable to try all possible orderings and choose the best one. However, there are some heuristic approaches that approximate the optimal solution. We follow the *iterative conditional mode* or ICM algorithm [5]. This is implemented as follows : at each iteration of ICM, we first randomly order the points. Then we assign the points (in that order) to the cluster i that minimizes $\mathcal{O}_{MCIKeans}(\boldsymbol{x})$. This is continued until no point changes its cluster in successive iterations, which indicates convergence. According to [5], ICM is guaranteed to converge. The E-step completes after termination of ICM, and the program moves to the M-step.

M-Step: In the M-Step, we re-compute each cluster centroid by averaging all the points in that cluster:

$$\boldsymbol{u_i} = \frac{\sum_{\boldsymbol{x} \in \mathcal{X}_i} \boldsymbol{x}}{|\mathcal{X}_i|} \tag{8}$$

After performing this step, the convergence condition is checked. If fulfilled, the procedure terminates, otherwise another iteration of E-Step and M-Step is performed.

4.4. Storing the classification model

After building the K clusters, we create a summary of the statistics of the data points belonging to each cluster. The summary contains the following statistics: i) N: the total number of points; ii) Lt: the total number of labeled points; iii) $\{Lp[c]\}_{c=1}^{C}$: a vector containing the total number of labeled points belonging to each class. iv) $\boldsymbol{u}$: the centroid of the cluster. v) $\hat{c}$: the majority class, i.e., the class having the highest frequency in the micro-cluster. This summary will be referred to henceforth as a “micro-cluster”. Note that with these statistics, the additive property of micro-clusters [1] remains valid. This property is essential for merging two micro-clusters. After creating the micro-clusters, we discard the raw data points. Besides, we also discard all micro-clusters that do not contain any labeled point (i.e., have $Lt = 0$) because these micro-clusters do not play any role in classification. The remaining set of micro-clusters serve as a classification model. Note that the number of micro-clusters in the model will become less than K if any such deletions take place.

5. Ensemble classification

The ensemble consists of L models, where each model is trained with a partially labeled data chunk according to section 4. The initial ensemble consists of the first L models trained with the first L chunks in the stream. The ensemble is used to classify future unlabeled instances. Besides, the ensemble undergoes several modifications in each successive chunk to keep it up-to-date with the most recent concept.

5.1. Classification

Classification is done using nearest neighbor technique. In order to classify an unlabeled data point $\boldsymbol{x}$ with a model M^i, we perform the following steps: i) find the nearest micro-cluster from $\boldsymbol{x}$ in M^i, by computing the distance between the point and the centroids of the micro-clusters. ii) select the class with the highest frequency of labeled instances as the predicted class of $\boldsymbol{x}$. Recall that the frequencies of labeled instances for each class are stored in the micro-cluster data structure. In order to classify $\boldsymbol{x}$ with the *ensemble* M, we perform the following steps: i) find the nearest micro-cluster from $\boldsymbol{x}$ in each model $M^i \in M$. Let the nearest micro-cluster from $\boldsymbol{x}$ in M^i be M^i_{min} ii) select the class with the highest "cumulative frequency" in these L micro-clusters $\{M^1_{min}, ..., M^L_{min}\}$ as the predicted class of $\boldsymbol{x}$. The classification by the ensemble can be thought of a kind of majority voting among all the voters (i.e., labeled points) in the L nearest micro-clusters $\{M^1_{min}, ..., M^L_{min}\}$.

As an example, suppose there are three models (M^1, M^2, M^3) in the ensemble (i.e., L=3), and C=2. The nearest micro-clusters from the test point $\boldsymbol{x}$ in M^1,M^2 and M^3 have the following class frequencies, respectively: [1,5],[3,1], and [0,3], where the first number in each pair represents the frequency of class c=1, and the second number represents the frequency of class c=2. Since the class-wise cumulative frequencies of the three micro-clusters are [1+3+0,5+1+3] = [4,9], the predicted label of $\boldsymbol{x}$ by the ensemble M will be $\hat{y} = 2$.

5.2. Ensemble refinement

After a new model M' has been trained with a partially labeled data chunk, the existing ensemble M is refined with this model (line 3, algorithm 1). Refinement is done if the latest partially labeled data chunk D_n contains a class c, which is absent in all models of the ensemble M. This is possible if either a completely new class appear in the stream or an old class re-appears that has been absent in the stream for a long time. Both of these happens because of concept-evolution, and the class c is denoted as an *evolved class*. Note that there may be more than one evolved classes in the stream. If there is any evolved class, M must be refined so that it can correctly classify future instances of that class. Algorithm 2 describes how the existing model is refined.

Description of "Refine-Ensemble" (algorithm 2): The algorithm starts (line 1) by checking whether ensemble refinement is needed. This can be done in constant time by keeping a boolean vector $\boldsymbol{V}$ of size C per model, and setting $V[c] = true$ during training if there is any labeled training instance from class c. The function Need-to-refine(M) checks whether there is any class c such that $V[c]$ is false for all models $M^i \in M$, but true for M'. If there is such a class c, then c is an evolved class. Refinement is needed only if there is an evolved class. Then the algorithm looks into each micro-cluster M'_j of the new model M' (line 2). If the majority class of M'_j is an evolved class (line 3), then we do the followings: for each model $M^i \in M$, we inject the micro-cluster M'_j in M^i (line 7). Before injecting a micro-cluster, we try to merge the closest pair of micro-

Algorithm 2 Refine-Ensemble

```
Input: $M$: current ensemble of $L$ models $\{M^1, ..., M^L\}$
    $M'$: the new model built from the new data chunk $D_n$
Output: Refined ensemble $M$
 1: if Need-to-refine($M$)=false then return $M$
 2: for each micro-cluster $M'_j \in M'$ do
 3:   if the majority class of $M'_j$ is an evolved class then
 4:     for each Model $M^i \in M$ do
 5:       $Q \leftarrow$ The closest pair of micro-clusters in $M^i$ having the same majority class
 6:       if $Q \neq null$ and $|M^i| = K$ then Merge the pair of micro-clusters in $Q$
 7:       $M^i \leftarrow M^i \cup M'_j$ /* Injection */
 8:     end for
 9:   end if
10: end for
11: return $M$
```

clusters in M^i having the same majority class (line 6). This is done to keep the number of micro-clusters constant (=K). However, merging is done only if such a closet pair is found, and $|M^i|$, the total number of micro-clusters in M^i equals K. Note that the first condition may occur (i.e., no such closest pair found) if $|M^i| < C$. In this case, $|M^i|$ is incremented after the injection. This ensures that if C, the number of classes, increases due to concept-evolution, the number of micro-clusters in each model also increases. In the extreme case (not shown in the algorithm) when C exceeds K due to evolution, K is also incremented to ensure that the relation $K > C$ remains valid. The reasoning behind the refinement is as follows. Since no model in ensemble M has knowledge of an evolved class c, the models will certainly misclassify any data belonging to the class. By injecting micro-clusters of the class c, we introduce some data from this class into the models, which reduces their misclassification rate.

It is obvious that when more training instances are provided to a model, its classification error is more likely to reduce. However, if the same set of micro-clusters are injected in all the models, the correlation among the models may increase, resulting in reduced prediction accuracy of the *ensemble*. According to [25], if the errors of the models in an L-model ensemble are independent, then the added error (i.e., the error in addition to Bayes error) of the ensemble is $1/L$ times the added error of a single model. However, the ensemble error may be higher if there is correlation among the errors of the models. But even if correlation is introduced by injecting the micro-clusters, according to the following lemma (lemma 1), under certain conditions the overall added error of the ensemble is reduced after injection. The lemma is based on the assumption that after injection, single model error monotonically decreases with increasing prior probability of class c. In other words, we assume that there is a continuous monotonic decreasing function $f(x)$, $f(x) \in [0, 1]$ and $x \in [0, 1]$, such that

$$\mathcal{E} = f(\gamma_c) * \mathcal{E}^0 \tag{9}$$

where $\mathcal{E}^0$ and $\mathcal{E}$ are the single model errors before and after injection, and γ_c is the prior probability of class c. This function has the following special property: f(0) = 1, since γ_c=0 means class c has not appeared at all, and no injection has been made. Lemma 1 quantifies an upper bound of the function that is necessary for ensemble error reduction.

Lemma 1. Let c be the evolved class, $\mathcal{E}^0_M$ and $\mathcal{E}_M$ be the added errors of the ensemble before and after injection; $\mathcal{E}^0$ and $\mathcal{E}$ be the added errors of a single model before and after injection, and γ_c be the prior probability of class c. Then the injection process will reduce the added error of the ensemble provided that

$$f(\gamma_c) \leq \frac{1}{1+\gamma_c^2(L-1)}.$$

where L is the ensemble size.

Proof: According to [25],

$$\mathcal{E}_M = \mathcal{E} * \frac{1+\delta(L-1)}{L} \tag{10}$$

where L is the total number of models in the ensemble, and δ is the mean correlation among the models, given by:

$$\delta = \sum_{i=1}^{C} \gamma_i \delta_i \tag{11}$$

where γ_i is the prior probability of class i and δ_i is the mean correlation associated with class i, given by [25]:

$$\delta_i = \frac{1}{L(L-1)} \sum_{m=1}^{L} \sum_{l \neq m} Corr(\eta_i^m, \eta_i^l) \tag{12}$$

where $Corr(\eta_i^m, \eta_i^l)$ is the correlation between η_i^m, the error of model m, and η_i^l, the error of model l. For simplicity, we assume that the correlation between two models is proportional to the number of instances that are common to both these models. That is, the correlation is 1 if they have all instances in common, and 0 if they have no instances in common. So, before injection, the correlation between any pair of models is zero (since the models are trained using disjoint training data). As a result,

$$\mathcal{E}_M^0 = \frac{\mathcal{E}^0}{L} \tag{13}$$

After injection, some instances of class c may be common among a pair of models, leading to $\delta_c \geq 0$, where c is the evolved class.

Consider a pair of models m and l whose prior probabilities of class c are γ_c^m and γ_c^l, respectively, after injection. So, the correlation between m and l reduces to:

$$Corr(\eta_c^m, \eta_c^l) = \frac{1}{2}(\gamma_c^m + \gamma_c^l)$$

Substituting this value in equation (12), we obtain

$$\begin{aligned} \delta_c =& \frac{1}{L(L-1)} \frac{1}{2} \sum_{m=1}^{L} \sum_{l \neq m} (\gamma_c^m + \gamma_c^l) \\ =& \frac{1}{L(L-1)} \frac{1}{2} 2(L-1) \sum_{m=1}^{L} (\gamma_c^m) = \frac{1}{L} \sum_{m=1}^{L} (\gamma_c^m) = \bar{\gamma_c} \end{aligned} \tag{14}$$

where $\bar{\gamma_c}$ is the mean prior probability of class c in each model. Note that the mean prior probability $\bar{\gamma_c}$ represents the actual prior probability γ_c, so they can be used interchangeably. Substituting this value of δ_i in equation (11),

$$\delta = \sum_{i=1}^{C} \gamma_i \delta_i = \gamma_c \delta_c + \sum_{i=1, i \neq c}^{C} \gamma_i \delta_i = (\gamma_c)^2 + 0 = (\gamma_c)^2$$

since $\delta_i = 0$ for all non-evolved class as no instance of those classes is common between any pair of models. Now, substituting this value of δ in equation (10), we obtain

$$\begin{aligned}
\mathcal{E}_M &= \mathcal{E} * \frac{1 + {\gamma_c}^2(L-1)}{L} \\
&= f(\gamma_c) * \mathcal{E}^0 * \frac{1 + {\gamma_c}^2(L-1)}{L} \text{ using (9)} \\
&= \frac{\mathcal{E}^0}{L} * (f(\gamma_c) * (1 + {\gamma_c}^2(L-1)) \\
&= \mathcal{E}_M^0 * (f(\gamma_c) * (1 + {\gamma_c}^2(L-1)) \text{ using (13)}
\end{aligned} \tag{15}$$

Now, we will have an error reduction provided that $\mathcal{E}_M \leq \mathcal{E}_M^0$, which leads to:

$$\begin{aligned}
&(f(\gamma_c) * (1 + \gamma_c^2(L-1)) \leq 1 \\
&f(\gamma_c) \leq \frac{1}{1 + \gamma_c^2(L-1)} \; \square
\end{aligned}$$

From lemma 1, we can infer that the function $f(.)$ becomes more restricted as the value of γ_c and/or L are increased. For example, for $\gamma_c = 0.5$, if L=10, then f(γ_c) must be ≤ 0.31, meaning, $\mathcal{E} \leq 0.31 * \mathcal{E}_0$ is required for error reduction. For the same value of γ_c, if L=2, then $\mathcal{E} \leq 0.8 * \mathcal{E}_0$ is required for error reduction. However, in our experiments, we have always observed error reduction after injection, i.e., inequality (15) has always been satisfied. Still, we recommend that the value of L be kept within 10 for minimizing the risk of violating inequality (15).

5.3. Ensemble update

After the refinement, the ensemble is updated to adapt to the concept-drift in the stream. This is done as follows. We have now L+1 models: L models from the ensemble and the newly trained model M'. One of these L+1 models is discarded, and the rest of them construct the new ensemble. The victim is chosen by evaluating the accuracy of each of these L+1 models on the labeled instances in the training data D_n. The model having the worst accuracy is discarded.

5.4. Time complexity

The ensemble training process consists of three main steps: 1) creating clusters using E-M, 2) refining the ensemble, and 3) updating the ensemble. Step 2) requires $O(KL)$ time, and step 3) requires $O(KLPS)$ time, where P is the proportion of labeled data ($P \leq 1$) in the chunk and S is the chunk-size. Step 1) (E-M) requires $O(KSI_{icm}I_{em})$ time, where I_{icm} is the average number of ICM iterations per E-step and I_{em} is the total number of E-M iterations. Although it is not possible to find the exact values of I_{icm} and I_{em} analytically, we obtain an approximation by observation. We observe from our experiments that I_{em} depend only on the chunk-size S, and I_{icm} is constant ($\approx$2) for any dataset. On average, a data chunk having 1000 instances requires 10 E-M iterations to converge. This increases sub-linearly with chunks-size. For example, a 2000 instance chunk requires 14 E-M iterations and so on. There are several reasons for this fast convergence of E-M, such as: 1) proportionate initial seed selection from the labeled data using farthest-fast traversal, and 2) using the compound impurity measure in the objective function. Therefore, the overall time-complexity of the ensemble training process of SmSCluster is $O(KS * (LP + g(S)))$, where g(.) is a sub-linear function. This complexity is almost linear in S for a moderate chunk-size. The time complexity of ensemble classification is $O(KLS)$, which is also linear in S for a fixed value of K and L.

6. Experiments

In this section we discuss the data sets used in the experiments, the system setup, and the results.

6.1. Dataset

We apply our technique on two synthetic and two real datasets. We generate two different kinds of synthetic datasets: concept-drifting, and concept-drifting with concept-evolving. The former dataset simulates only concept-drift, whereas the latter simulates both concept-drift and concept-evolution. One of the two real datasets is the 10% version of the KDD cup 1999 intrusion detection dataset [19]. The other one is the Aviation Safety Reporting Systems (ASRS) dataset obtained from NASA [23]. All of these datasets are discussed in the following paragraphs.

Concept-drifting synthetic dataset (SynD): We use this dataset in order to show that our approach can handle concept-drift. SynD data are generated using a moving hyperplane technique. The equation of a hyperplane is as follows:

$$\sum_{i=1}^{d} a_i x_i = a_0.$$

where d is the total number of dimensions, a_i is the weight associated with dimension i, and x_i is the value of ith dimension of a datapoint x. If $\sum_{i=1}^{d} a_i x_i \leq a_0$, then an example is considered as negative, otherwise it is considered positive. Each instance is a randomly generated d-dimensional vector $\{x_1, ..., x_d\}$, where $x_i \in [0, 1]$. Weights $\{a_1, ..., a_d\}$ are also randomly initialized with a real number in the range [0, 1]. The value of a_0 is adjusted so that roughly the same number of positive and negative examples are generated. This can be done by choosing $a_0 = \frac{1}{2}\sum_{i=1}^{d} a_i$. We also introduce noise randomly by switching the labels of p% of the examples, where p=5 is set in our experiments.

There are several parameters that simulate concept drift. Parameter m specifies the percent of total dimensions whose weights are involved in changing, and it is set to 20%. Parameter t specifies the magnitude of the change in every N

examples. In our experiments, t is varied from 0.1 to 1.0, and N is set to 1000. $s_i, i \in \{1, ..., d\}$ specifies the direction of change for each weight. Weights change continuously, i.e., a_i is adjusted by $s_i.t/N$ after each example is generated. There is a possibility of $r\%$ that the change would reverse direction after every N examples are generated. In our experiments, r is set to 10%. We generate a total of 250,000 instances and divide them into equal-sized chunks.

Concept-drifting with Concept-evolving synthetic dataset (SynDE): SynDE dataset simulates both concept-drift and concept-evolution. That is, new classes appear in the stream as well as old classes disappear, and at the same time, the concept for each class gradually changes over time. The dataset size is varied from 100K to 1000K points. Number of class labels are varied from 5 to 40, and data dimensions are varied from 20 to 80. Data points belonging to each class are generated by following a Normal distribution having different mean (-5.0 to +5.0) and variance (0.5 to 6) for different classes. In order to simulate the evolving nature of data streams, the prior probabilities of different classes are varied with time. This has caused some classes to appear and some other classes to disappear at different times in the stream history. In order to simulate the drifting nature of the concepts, the class mean for each class are gradually changed in a way similar to the Syn-D dataset. Different synthetic datasets are identified by an abbreviation: <size> C <#of classes> D <#of dimensions>. For example, 300KC5D20 denotes a dataset having 300K points, 5 classes and 20 dimensions.

Real dataset-KDDCup 99 network intrusion detection (KDD): This dataset contains TCP connection records extracted from LAN network traffic at MIT Lincoln Labs over a period of two weeks. We have used the 10% version of the dataset, which is more concentrated than the full version. Here different classes appear and disappear frequently. Each instance in the dataset refers to either to a normal connection or an attack. There are 22 types of attacks, such as buffer-overflow, portsweep, guess-passwd, neptune, rootkit, smurf, spy, etc. So, there are 23 different classes of data, Most which are normal. Each record consists of 42 attributes, such as connection duration, the number bytes transmitted, number of root accesses, etc. we use only the 34 continuous attributes, and remove the categorical attributes.

Real dataset-Aviation Safety Reporting Systems (ASRS): This dataset contains around 150,000 text documents. Each document is actually a report corresponding to a flight anomaly. There are a total of 55 anomalies, such as "aircraft equipment problem : critical", "aircraft equipment problem : less severe", "inflight encounter : birds", "inflight encounter : skydivers", "maintenance problem : improper documentation" etc. Each of these anomalies is considered as a "class". These documents represent a data stream since it contains the reports in order of their creation time, and new reports are being added to the dataset on a regular basis.

We perform several preprocessing steps on this dataset. First, we discard the classes that contain very few (less than 100) documents. We choose 21 classes among the 55, which reduced the total number of selected documents to 125,799. Second, we extract word features from this corpus, and select the best 1000 features based on information gain. Then each document is transformed into a binary feature vector, where the value corresponding to a feature is "one" if the feature (i.e., word) is present, or “zero” if it is not present in the document. The instances in dataset are multi-label, meaning, an instance may have more than one class label. We transform the multi-label classification problem into 21 separate binary classification problems by generating 21 different datasets from the original dataset, one for each class. The dataset for i-th class is generated by marking the instances belonging to class i as positive, and all other instances as negative. When reporting the accuracy, we report the average accuracy of the 21 datasets.

6.2. Experimental setup

Hardware and software: We implement the algorithms in Java. The experiments were run on a Windows-based Intel P-IV machine with 2GB memory and 3GHz dual processor CPU.

Parameter settings: The default parameter settings are as follows, unless mentioned otherwise:
i) K (number of micro-clusters) = 50 for all datasets;
ii) S (chunk size) = 1,600 records for real datasets, and 1,000 records for synthetic datasets;
iii) L (ensemble size) = 10 for all datasets;

Baseline method: We compare our algorithm with “On Demand Stream”, propsoed by Aggarwal et al [1]. We will refer to this approach as “OnDS”. We run our own implementation of the OnDS and report the results. For the OnDS, we use all the default values of its parameters, and set buffer-size = 1,600 and stream-speed = 80 for real datasets, and buffer-size = 1,000 and stream-speed = 200 for synthetic datasets, as proposed by the authors. However, in order to ensure a fair comparison, we make a small modification to the original OnDS algorithm. The original algorithm assumed that in each data chunk, 50% of the instances are labeled, and the rest of them are unlabeled. The labeled instances were used for training, and the unlabeled instances are used for testing and validation. As mentioned earlier, this assumption is even more impractical than assuming that a single stream contains both training and test instances. Therefore, in the modified algorithm, we assume that all the instances in a new data chunk are unlabeled, and test all of them using the existing model. After testing, the data chunk is assumed to be completely labeled, and all the instances are used for training.

When training ReaSC, we consider that only 20% randomly chosen instances in a chunk have labels (i.e., P=20), whereas for training OnDS, 100% instances in the chunk are assumed to have labels. So, if there are *100 data points in a chunk, then OnDS has 100 labeled training data points, but ReaSC has only 20 labeled and 80 unlabeled training instances.* Also, for a fair comparison, the chunk-size of ReaSC is always kept equal to the buffer size of OnDS. Note that P is not a parameter of ReaSC, rather, it is a threshold assigned by the user based on the available system resources to label data points.

Evaluation: For each competing approach, we use the first 3 chunks to build the initial classification model, which can be thought of as an warm-up period. From the 4th chunk onward, we first evaluate the classification accuracy of the

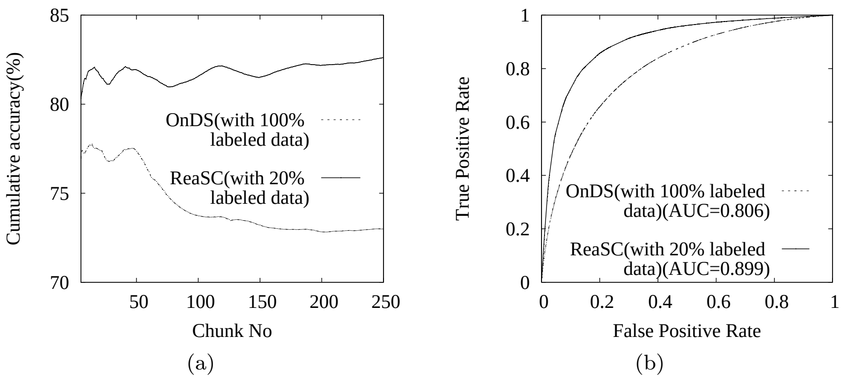


Fig. 2. Cumulative accuracy (a) and ROC curve (b) for SynD dataset

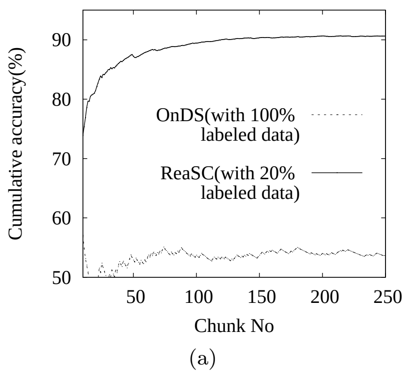
model on that chunk, then use the chunk as training data to update the model. Each method is run 20 times on each dataset, and the average result is reported.

6.3. Comparison with baseline methods

Figures 2(a)-5(b) compare the accuracies and receiver operating characteristic (ROC) curves for each dataset. Each of these graphs is generated by averaging 20 runs for each method for the same parameter settings.

Figure 2(a) shows the cumulative accuracy of each competing method for each chunk on SynD dataset. In this figure, the X-axis represents chunk number and the Y-axis represents accuracy of a particular method from the beginning of the stream. For example, in figure 2(a) at chunk 250 (X=250), the Y values for ReaSC and OnDS represent the cumulative accuracies of ReaSC and OnDS from the beginning of the stream to chunk 250, which are 82.61% and 73%, respectively. This curve shows that as the stream progresses, accuracy of OnDS declines. This is because OnDS is not capable of handling concept-drift properly. Figure 2(b) shows the ROC curve for SynD dataset. The ROC curve is a good visual representation of the overall performance of a classifier in classifying all the classes correctly. Sometimes only the accuracy measure does not properly reflect the true classification performance if the class distribution is skewed. ROC curves reflect the true performance even if the class distributions are skewed. The area under the ROC curve (AUC) is higher for a better classifier. The AUCs for each ROC curve is reported in each graph. For the SynD dataset, AUC of ReaSC is almost 10% higher than that of OnDS.

Figure 3(a) shows the chunk number versus cumulative accuracy for SynDE dataset. In this dataset, ReaSC performs better (90%) than SynD because SynDE is generated using Gaussian distribution, which is easier to learn for ReaSC. On the other hand, accuracy of OnDS is much worse in this dataset. In fact, the average accuracy of OnDS is always less than 55%. Recall that SynDE simulates both concept-drift and concept-evolution. Since OnDS performance poorly in a dataset having only concept-drift, it is natural that it performs even poorer in a dataset that has an additional hurdle: concept-evolution. The ROC of ReaSC on this dataset shown in figure 3(b) also has more than 20% higher AUC than OnDS.



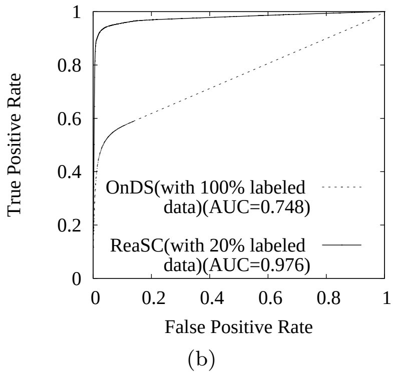
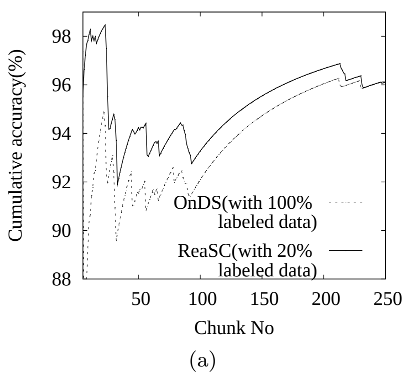


Fig. 3. Cumulative accuracy (a) and ROC curve (b) for SynDE dataset



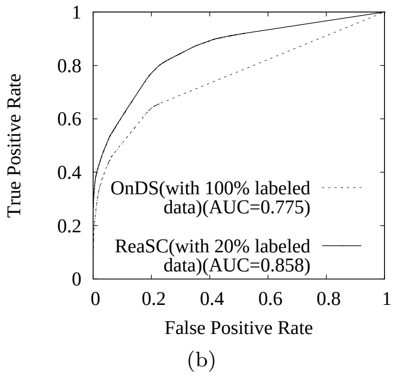
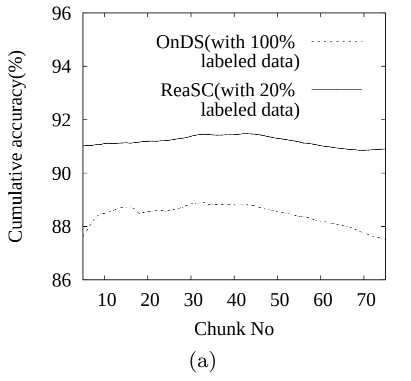


Fig. 4. Cumulative accuracy (a) and ROC curve (b) for KDD dataset



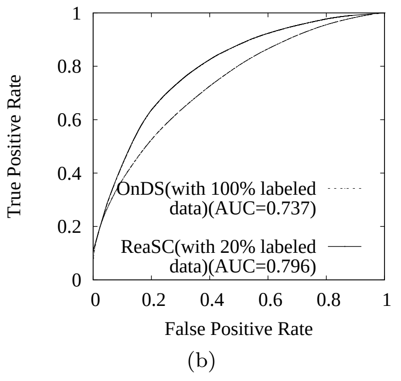


Fig. 5. Cumulative accuracy (a) and ROC curve (b) for ASRS dataset

Table 3. Comparison of running time (excluding labeling time) and classification speed between OnDS (with 100% labeled data) and ReaSC (with 20% labeled data)

DataSet	**Time (sec/1000 pts)**		**Classification speed (pts/sec)**	
	OnDS (100% labeled)	ReaSC (20% labeled)	OnDS (100% labeled)	ReaSC (20% labeled)
SynD	**0.88**	1.34	1,222	**6,248**
SynDE	**1.57**	1.72	710	**4,033**
KDD	1.54	**1.32**	704	**3,677**
ASRS	30.90	**10.66**	38	**369**

Figures 4(a) and 4(b) show the chunk no vs. cumulative accuracy and ROC curves for KDD dataset. KDD dataset has a lot of concept-evolution, almost all of which occur within the first 120 chunks. The accuracy of OnDS is 2-12% lower than ReaSC in this region. So, ReaSC handles concept-evolution better than OnDS in real data too. However, in KDD dataset, most of the instances belong to the “normal” class. As a result, the class distribution is skewed, and simple accuracy does not reflect the true difference in performances. The ROC curves shown in 4(b) reflects the performances of these two methods more precisely. The AUC of ReaSC is found to be 10% higher than OnDS, which is a great improvement. Finally, figures 5(a) and 5(b) show the accuracy and ROC curves for ASRS dataset. Recall that these graphs are generated by averaging the accuracies and ROC curves from 21 individual binary classification results. Again, here ReaSC achieves 3% or higher accuracy than OnDS in all stream positions. Besides, the AUC of ReaSC in this dataset is 8% higher than OnDS. OnDS performs comparatively better in this dataset because this dataset does not have any concept-drift.

Again recall that in all these experiments, OnDS uses 5 times more labeled data for training than ReaSC, still ReaSC outperforms OnDS in all datasets, both in accuracy and AUC.

6.4. Running times, scalability, and memory requirement

Table 3 compares the running times and classification speeds between ReaSC and OnDS. The columns headed by “Time (sec/1000 pts)” report the total running times (training plus testing) in *seconds per thousand points* of each of these methods. Note that these running times do not consider the data labeling time, which is an essential part of classifier training, and is a major bottleneck for OnDS, to be explained shortly. The columns headed by “classification speed (pts/sec)” report classification speed of each of these methods in *points per second*. The total running times of ReaSC in synthetic datasets are slightly higher than OnDS, but lower in real datasets. It is worth mentioning that the dimensions of the datasets are in increasing order : (SynD=10, SynDE=20, KDD=34, ASRS=1000), so are the running times. Both OnDS and ReaSC appear to have linear growth of running time with increasing dimensionality and class labels. But the running time of OnDS certainly grows at a higher rate than that of ReaSC with the increasing number of dimensions and class labels, as suggested by the the data presented in table 3. This is because, there is a classification overhead associated with OnDS, which increases with both stream length, data dimension and class labels, but there is no such overhead with ReaSC. The reason is that OnDS keeps snapshots of the micro-clusters for different time-stamps

Table 4. Comparison of running time including labeling time for real datasets

DataSet	**Labeling time (sec/1000 pts)**		**Total time (sec/1000 pts)**	
	OnDS (100% labeled)	ReaSC (20% labeled)	OnDS (100% labeled)	ReaSC (20% labeled)
KDD	1,000	200	1,001.54	**201.32**
ASRS	60,000	12,000	60,030.92	**12,010.66**

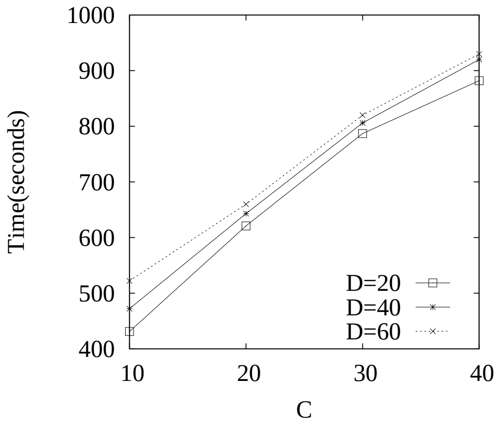


Fig. 6. Running times on different datasets having higher dimensions (D) and number of classes (C)

in stream history. When classification is needed, OnDS needs to find the best time horizon by searching through the saved snapshots. This searching time is directly related with the data dimension, number of class labels, and stream length. As a result, OnDS takes relatively higher time on higher dimensions and larger datasets than ReaSC. As also shown in the table, classification speed of OnDS is much lower than ReaSC for the same reason.

If we include data labeling time, we get a more real picture of the total running time. Suppose the labeling time for each data point for KDD dataset is 1 sec, and the same for ASRS dataset is 60 seconds. In fact, real annotation times would be much higher for any text dataset [27]. Table 4 shows the comparison. The labeling time for OnDS is 5 times higher than that of ReaSC, since per 1,000 instances OnDS requires 1,000 instances to have label, whereas ReaSC requires only 200 instances to have label. The net effect is, ReaSC is 5 times faster than OnDS in both datasets.

In figure 6, we report the scalability of ReaSC on high-dimensional and multi-class SynDE data. This graph reports the running times of ReaSC for different dimensions (20-60) of synthetic data with different number of classes (10-40). Each of these synthetic datasets has 250K points. For example, for C=10, and D=20, the running time is 431 seconds, and it increases linearly with the number of classes in the data. On the other hand, for a particular value of C (e.g. D=20), the running time increases very slowly (linearly) with increasing the number of dimensions in the data. For example, for C=10, running times for 20, 40, and 60 dimensions of datasets are 431, 472, and 522 seconds, respectively. Thus, we may conclude that ReaSC scales linearly to higher dimensionality and class labels.

The memory requirement for ReaSC is $O(D*K*L)$, whereas that of OnDS is

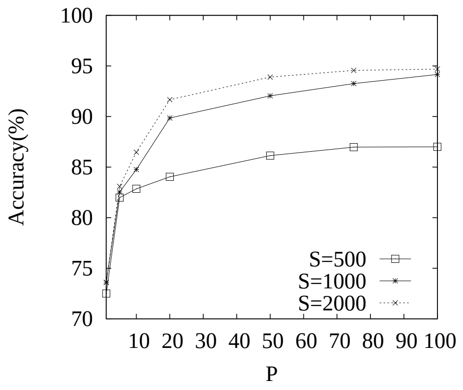


Fig. 7. Sensitivity to chunk size (S) for different percentage of labeled data (P)

$O(D{*}microcluster\_ratio{*}max\_capacity{*}C{*}log(N))$, where N is the total length of the stream. Thus, the memory requirement of ReaSC is constant, whereas that of OnDS grows with stream length. For high dimensional datasets, this requirement may not be practical. For example, for the ASRS dataset, ReaSC requires less than 10MB memory, whereas OnDS requires approximately 700MB memory.

6.5. Sensitivity to parameters

All the following results are obtained using a SynDE dataset (B250K,C10,D20). Figure 7 shows how accuracy varies with chunk size (S) and the percentage of labeled instances in each chunk (P). It is obvious that higher values of P leads to better classification accuracy since each model is better trained. For any particular chunk size, the improvement gradually diminishes as P approaches to 100. For example, a stream with P=10 has 5 times more labeled data than the one with P=2. As a result, the accuracy improvement is also rapid from P=2 to P=10. But a stream with P=75 has only 1.5 times more labeled data than a stream with P=50, so the accuracy improvement in this case is much less than the former case. We also observe higher accuracy for larger chunk sizes. This is because, as chunk size is increased, each model gets trained with more data, which leads to a better classification accuracy. This improvement also diminishes gradually because of concept-drift. According to [29], if there is concept-drift in the data, then a larger chunk contains more outdated points, canceling out any improvement expected to be gained by increasing the training set size.

Figure 8(a) shows how classification accuracy varies for ReaSC with the number of micro-clusters (K). We observe that higher values of K lead to better classification accuracies. This happens because when K is larger, smaller and more compact clusters are formed, leading to a finer-grained classification model for the nearest neighbor classifier. However, there is no significant improvement after K=50 for this dataset, where C=10. It should be noted that K should always be much larger than C. Experimental results suggests that K should be between $2C$ and $5C$ for best performance.

Figure 8(b) shows the effect of accuracy on ensemble size (L). Intuitively,

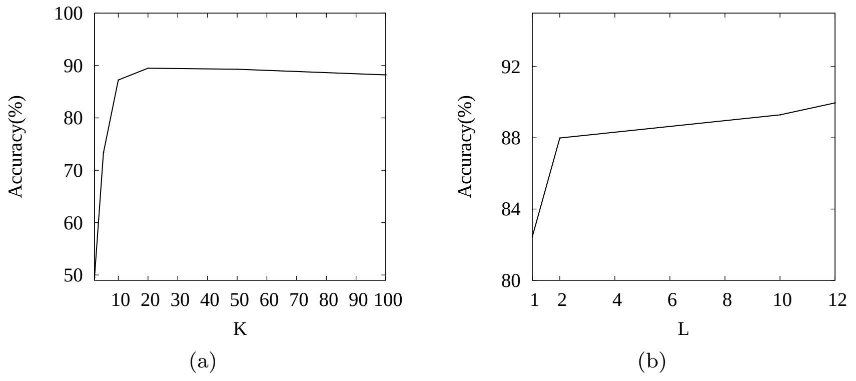


Fig. 8. Sensitivity to number of clusters (K) (a) and ensemble size (L) (b)

increasing the ensemble size helps to reduce error. Significant improvement is achieved by increasing the ensemble size from 1 (i.e., single classifier) to 2. After that, the improvement diminishes gradually. Increasing the ensemble size also increases the classification time. Besides, correlation among the classifiers increases in the event of concept-evolution, which diminishes the improvement intended by the ensemble. So, a reasonable value is chosen depending on the specific requirements of a system.

7. Discussion

From the above results, we can conclude that ReaSC outperforms OnDS in all datasets. There are two main reasons behind this. First, ReaSC considers both the dispersion and impurity measures in building clusters, but OnDS considers only purity, since it applies K-means algorithm to each class separately. Besides, ReaSC uses proportionate initialization, so that more clusters are formed for the larger classes (i.e., classes having more instances). But OnDS builds equal number of clusters for each class, so clusters belonging to larger classes tend to be bigger (and more sparse). Thus, the clusters of ReaSC are likely to be more compact than those of the OnDS. As a result, the nearest neighbor classification gives better prediction accuracy in ReaSC. Second, ReaSC applies ensemble classification, rather than the "horizon fitting" technique used in OnDS. Horizon fitting selects a horizon of training data from the stream that corresponds to a variable-length window of the most recent (contiguous) data chunks. It is possible that one or more chunks in that window have been outdated, resulting in a less accurate classification model. This is because the set of training data that is the best representative of the current concept are not necessarily contiguous. But ReaSC always keeps the best training data (or models) that are not necessarily contiguous. So, the ensemble approach is more flexible in retaining the most up-to-date set of training data, resulting in a more accurate classification model.

It would be interesting to compare ReaSC with some other baseline approaches. First, consider a *single* combined model that contains all the $K * L$ clusters in the ensemble M. We argue that this combined model is no better than the ensemble of models because our analysis shows that increasing the number of clusters beyond a certain threshold (e.g. 100) does not improve classification accuracy. Since K is chosen to be close to this threshold, it is most likely that we would not get a better model out of the $K * L$ clusters. Second, consider a single model having K clusters (not exceeding the threshold) built from L data chunks. Increasing the training set size would most likely improve classification accuracy. However, in the presence of concept drift, it can be shown that a single model built from L consecutive data chunks has a prediction error no less than an ensemble of L models, each built on a single data chunk [29]. This also follows from our experimental results that a single model built on L chunks has 5%-10% worse accuracy than ReaSC, and is at least L-times slower than ReaSC.

8. Conclusion

We address a more realistic problem of stream mining: training with a limited amount of labeled data. Our technique is a more practical approach to the stream classification problem since it requires a fewer amount of labeled data, saving much time and cost that would be otherwise required to manually label the data. Previous approaches for stream classification did not address this vital problem.

We propose and implement a semi-supervised clustering based stream classification algorithm to solve this limited labeled-data problem. We show that our approach, using much fewer labeled training instances than other stream classification techniques, works better than those techniques. We evaluated our technique on two synthetically generated datasets, and two real datasets, and achieved better classification accuracies than state-of-the-art stream classification approaches in all datasets.

In future, we would like to incorporate feature-weighting and distance-learning in the semi-supervised clustering, which should lead to a better classification model. Besides, we would like to apply our technique to classify other real stream data.

Acknowledgment

This research was funded in part by NASA grant, NNX08AC35A.

References

[1] C. C. Aggarwal, J. Han, J. Wang, and P. S. Yu. A framework for on-demand classification of evolving data streams. *IEEE Transactions on Knowledge and Data Engineering*, 18(5):577–589, 2006.

[2] S. Basu, A. Banerjee, and R. J. Mooney. Semi-supervised clustering by seeding. In *Proc. Nineteenth International Conference on Machine Learning (ICML)*, pages 19–26, Sydney, Australia, July 2002.

[3] S. Basu, A. Banerjee, and R. J. Mooney. Active semi-supervision for pairwise constrained clustering. In *Proc. SIAM International Conference on Data Mining (SDM)*, pages 333–344, Lake Buena Vista, FL, April 2004.

[4] S. Basu, M. Bilenko, A. Banerjee, and R. J. Mooney. Probabilistic semi-supervised clustering with constraints. *Semi-Supervised Learning, O. Chapelle, B. Schoelkopf, and A. Zien (eds)*, pages 73–102, 2006.

[5] J. Besag. On the statistical analysis of dirty pictures. *Journal of the Royal Statistical Society, Series B (Methodological)*, 48(3):259–302, 1986.

[6] M. Bilenko, S. Basu, and R. J. Mooney. Integrating constraints and metric learning in semi-supervised clustering. In *Proc. 21st International Conference on Machine Learning (ICML)*, pages 81–88, Banff, Canada, July 2004.

[7] S. Chen, H. Wang, S. Zhou, and P. Yu. Stop chasing trends: Discovering high order models in evolving data. In *Proc. ICDE*, pages 923–932, 2008.

[8] D. Cohn, R. Caruana, and A. McCallum. Semi-supervised clustering with user feedback. *Technical Report TR2003-1892, Cornell University*, 2003.

[9] A. Demiriz, K. P. Bennett, and M. J. Embrechts. Semi-supervised clustering using genetic algorithms. In *Artificial Neural Networks in Engineering (ANNIE-99)*, pages 809–814. ASME Press, 1999.

[10] A. P. Dempster, N. M. Laird, and D. B. Rubin. Maximum likelihood from incomplete data via the em algorithm. *Journal of the Royal Statistical Society B*, 39:1–38, 1977.

[11] P. Domingos and G. Hulten. Mining high-speed data streams. In *Proc. ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*, pages 71–80, Boston, MA, USA, 2000. ACM Press.

[12] W. Fan. Systematic data selection to mine concept-drifting data streams. In *Proc. ACM SIGKDD International Conference on Knowledge Discovery and Data Mining (KDD)*, pages 128–137, Seattle, WA, USA, 2004.

[13] Y. Freund and R. E. Schapire. Experiments with a new boosting algorithm. In *In Proceedings of the Thirteenth International Conference on Machine Learning (ICML)*, pages 148–156, Bari, Italy, 1996. Morgan Kaufmann.

[14] J. Gao, W. Fan, and J. Han. On appropriate assumptions to mine data streams. In *Proc. Seventh IEEE International Conference on Data Mining (ICDM)*, pages 143–152, Omaha, NE, USA, Oct 2007.

[15] J. Gehrke, V. Ganti, R. Ramakrishnan, and W. Loh. Boat-optimistic decision tree construction. In *Proc. ACM SIGMOD international conference on Management of data (SIGMOD)*, pages 169–180, Philadelphia, PA, USA, May 1999.

[16] M. Halkidi, D. Gunopulos, N. Kumar, M. Vazirgiannis, and C. Domeniconi. A framework for semi-supervised learning based on subjective and objective clustering criteria. In *Proc. Fifth IEEE International Conference on Data Mining (ICDM)*, pages 637—640, Houston, Texas, USA, Nov 2005.

[17] D. Hochbaum and D. Shmoys. A best possible heuristic for the k-center problem. *Mathematics of Operations Research*, 10(2):180–184, 1985.

[18] G. Hulten, L. Spencer, and P. Domingos. Mining time-changing data streams. In *Proc. seventh ACM SIGKDD international conference on Knowledge discovery and data mining (KDD)*, pages 97–106, San Francisco, CA, USA, Aug 2001.

[19] KDD Cup 1999 Intrusion Detection Dataset. `http://kdd.ics.uci.edu/databases/kddcup99/kddcup99.html`.

[20] D. Klein, S. D. Kamvar, and C. D. Manning. From instance-level constraints to space-level constraints: Making the most of prior knowledge in data clustering. In *Proc. 19th International Conference on Machine Learning (ICML)*, pages 307–314, Sydney, Australia, July 2002. Morgan Kaufmann Publishers Inc.

[21] J. Kolter and M. Maloof. Using additive expert ensembles to cope with concept drift. In *Proc. International Conference on Machine Learning (ICML)*, pages 449–456, Bonn, Germany, Aug 2005.

[22] M. M. Masud, J. Gao, L. Khan, J. Han, and B. Thuraisingham. A practical approach to classify evolving data streams: Training with limited amount of labeled data. In *Proc. International Conference on Data Mining (ICDM)*, Pisa, Italy, Dec 15-19 2008.

[23] NASA Aviation Safety Reporting System. `http://akama.arc.nasa.gov/ASRSDBOnline/QueryWizard_Begin.aspx`.

[24] M. Scholz and R. Klinkenberg. An ensemble classifier for drifting concepts. In *Proc. Second International Workshop on Knowledge Discovery in Data Streams (IWKDDS)*, pages 53–64, Porto, Portugal, Oct 2005.

[25] K. Tumer and J. Ghosh. Error correlation and error reduction in ensemble classifiers. *Connection Science*, 8(304):385–403, 1996.

[26] P. E. Utgoff. Incremental induction of decision trees. *Machine Learning*, 4:161–186, 1989.

[27] G. B. van Huyssteen, M. J. Puttkammer, S. Pilon, and H. J. Groenewald. Using machine learning to annotate data for nlp tasks semi-automatically. In *Proc. Computer-Aided Language Processing (CALP'07)*, 2007.

[28]K. Wagsta, C. Cardie, and S. Schroedl. Constrained k-means clustering with background knowledge. In *Proc. 18th International Conf. on Machine Learning (ICML)*, pages 577–584, Williamstown, MA, USA, June 2001. Morgan Kaufmann.

[29]H. Wang, W. Fan, P. S. Yu, and J. Han. Mining concept-drifting data streams using ensemble classifiers. In *Proc. ninth ACM SIGKDD international conference on Knowledge discovery and data mining*, pages 226–235, Washington, DC, USA, Aug, 2003. ACM.

[30]E. P. Xing, A. Y. Ng, M. I. Jordan, and S. Russell. Distance metric learning, with application to clustering with side-information. In *Advances in Neural Information Processing Systems 15*, pages 505–512. MIT Press, 2003.

[31]Y. Yang, X. Wu, and X. Zhu. Combining proactive and reactive predictions for data streams. In *Proc. KDD*, pages 710–715, 2005.